UCRL-MA-145712

User Documentation for Sensida, A Variant of IDA for Sensitivity Analysis

S. L. Lee, A. C. Hindmarsh

October 8, 2001

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

USER DOCUMENTATION FOR SENSIDA, A VARIANT OF IDA FOR SENSITIVITY ANALYSIS*

STEVEN L. LEE AND ALAN C. HINDMARSH †

1. Introduction. SensIDA and IDA [9] are general-purpose codes for solving differential-algebraic equation (DAE) initial value problems. SensIDA is a variant of IDA that includes options for simultaneously computing the DAE solution together with its first-order sensitivity coefficients with respect to model parameters. SensIDA is written in ANSI-standard C and it is mainly based on IDA, DASPK3.0 [11], and SensPVODE [10]. IDA is based on DASPK2.0 [3]. DASPK3.0 is a Fortran77 code for the sensitivity analysis of DAE initial value problems. SensPVODE is a sensitivity analysis variant of the parallel ordinary differential equation solver PVODE [5].

SensIDA can be compiled to run on serial or parallel computers. This is accomplished by specifying that the serial or parallel version of the vector module `NVECTOR` is used when compiling SensIDA. The parallel version of SensIDA uses MPI (Message-Passing Interface [8]) to achieve parallelism, and is intended for a distributed Single Program Multiple Data environment in which all vectors are identically partitioned across processors. The idea is for each processor to solve a certain fixed subset of the DAEs that describe the model problem and the first-order sensitivity coefficients of the solution.

SensIDA includes all of the numerical methods contained in IDA: backward differentiation formulas (BDF) for time integration: Newton/direct methods or an Inexact Newton/Krylov method for solving the nonlinear systems: preconditioning modules for Krylov subspace methods such as GMRES [13]. The linear solver and preconditioning modules allow for other direct methods, Krylov methods, and user-supplied preconditioners to be easily included. SensIDA also retains the use of matrix-free methods [2] for approximating preconditioned matrix-vector products. This approach obtains matrix-vector products within GMRES without explicitly computing or storing the linear system matrix. The parallel version of SensIDA includes only the Krylov method GMRES for solving linear systems. SensIDA was developed and tested on a cluster of Sun-SPARC workstations.

The remainder of this document is organized as follows: Section 2 sets the mathematical notation and summarizes the basic approach to sensitivity analysis. Section 3 summarizes the organization of the SensIDA solver, while Section 4 summarizes its usage. Section 5 describes three example problems. Finally, Section 6 discusses the availability of SensIDA.

2. Mathematical Considerations. In many large-scale computational simulations, the governing equations can often be spatially discretized and then numerically solved as a system of DAEs. Typically, these equations contain parameter values (for example, reaction rates and problem coefficents) that are not precisely known. In addition to numerically

* This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

† Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551

solving the DAEs, it may be desirable to determine the sensitivity of the results with respect to the model parameters. Such sensitivity information is useful because it indicates which parameters are most influential in affecting the behavior of the simulation.

SensIDA is a variant of IDA that computes the first-order sensitivity of the DAE solution with respect to some or all of its model parameters. When computing sensitivities in this context, one is interested in solving the DAE initial value problem

$$F(t, y, y', p) = 0 \tag{1}$$

where y, y' and F are vectors in $\mathbf{R}^N$, p is a vector of parameters in $\mathbf{R}^m$, and t is the independent variable. The first-order *sensitivities* are defined as

$$s_i(t) = \frac{\partial y(t,p)}{\partial p_i}, \ i = 1, \cdots, m,$$

and the sensitivity equations are obtained by differentiating (1) with respect to each parameter p_i:

$$\frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial y'} s_i' + \frac{\partial F}{\partial p_i} = 0, \ i = 1, \cdots, m. \tag{2}$$

If $\overline{p}_i$ is a nonzero constant, equations for the *scaled* sensitivities

$$w_i(t) = \overline{p}_i s_i(t) \tag{3}$$

can be obtained by multiplying each equation (2) by $\overline{p}_i$:

$$\frac{\partial F}{\partial y} w_i + \frac{\partial F}{\partial y'} w_i' + \overline{p}_i \frac{\partial F}{\partial p_i} = 0, \ i = 1, \cdots, m. \tag{4}$$

This scaling is done in order to obtain vectors w_i with the same physical units as the components in y.

SensIDA carries out the time integration of the combined system (1) and (4) by viewing it as a DAE system of size $N(m+1)$. By defining

$$Y(t) = \begin{pmatrix} y(t) \\ w_1(t) \\ \vdots \\ w_m(t) \end{pmatrix} \quad \text{and} \quad \mathbf{F}(t, Y, Y', p) = \begin{pmatrix} F(t, y, y', p) \\ \frac{\partial F}{\partial y} w_1(t) + \frac{\partial F}{\partial y'} w_1'(t) + \overline{p}_1 \frac{\partial F}{\partial p_1} \\ \vdots \\ \frac{\partial F}{\partial y} w_m(t) + \frac{\partial F}{\partial y'} w_m'(t) + \overline{p}_m \frac{\partial F}{\partial p_m} \end{pmatrix},$$

the combined DAE system is simply

$$\mathbf{F}(t, Y, Y', p) = 0 \tag{5}$$

and the initial conditions

$$Y(t_0) = Y_0(p), \quad Y'(t_0) = Y_0'(p)$$

must be consistent so that they satisfy (5). Given initial guesses for $Y(t_0)$ and $Y'(t_0)$, SensIDA includes a routine that computes consistent initial conditions for certain classes of DAE problems [4].

SensIDA uses backward differentiation formulas (BDFs) of order 1 through 5 to integrate the system (5). The first-order case is the backward Euler method, and in that case this approach yields the nonlinear system

$$\mathbf{0} = G(Y_{n+1}) \equiv \mathbf{F}(t_{n+1}, Y_{n+1}, \frac{Y_{n+1} - Y_n}{h}, p) \tag{6}$$

where $h = t_{n+1} - t_n$ is the current stepsize. Due to the form of $\mathbf{F}$, the Jacobian matrix $\partial G/\partial Y$ has the lower block triangular structure

$$\frac{\partial G}{\partial Y} = \begin{pmatrix} J & & \\ J_1 & J & \\ \vdots & & \ddots \\ J_m & & & J \end{pmatrix}$$

where

$$J = \frac{1}{h}\frac{\partial F}{\partial y'} + \frac{\partial F}{\partial y} \quad \text{and} \quad J_i = \frac{\partial}{\partial y}\left(\frac{\partial F}{\partial y}w_i + \frac{\partial F}{\partial y'}w_i' + \overline{p}_i\frac{\partial F}{\partial p_i}\right).$$

Higher-order BDFs also yield Jacobian matrices $\partial G/\partial Y$ with this same lower block triangular structure and with identical block-diagonal entries, J. The only difference is that the coefficient $1/h$ becomes α_0/h, where α_0 is a coefficient in the BDF. SensIDA solves the nonlinear systems $G(Y_{n+1}) = \mathbf{0}$ by using the simultaneous corrector method [12], a technique in which the Newton iteration uses the block-diagonal portion of $\partial G/\partial Y$ as the linear system matrix. This results in a decoupling that allows J to be used repeatedly in solving the $(m+1)$ linear systems that arise: one linear system for the Newton correction to the state variables y, and m linear systems for the corrections to the m scaled sensitivity vectors w_i. Because all of the Jacobian matrices are identical, the latter systems are solved using the same preconditioner and/or linear system solver that were specified for the original DAE problem (1).

The integrator computes an estimate E_n of the local error at each time step, and strives to satisfy the inequality

$$\|E_n\|_{rms,W} < 1 \ . \tag{7}$$

Here the weighted root-mean-square (rms) norm is defined by

$$\|E_n\|_{rms,W} = \left[\sum_{i=1}^{N(m+1)} \frac{1}{N(m+1)}\left(W_i E_{n,i}\right)^2\right]^{1/2} \tag{8}$$

where $E_{n,i}$ denotes the ith component of E_n, and the ith component of the weight vector is

$$W_i = \frac{1}{\text{RTOL}|Y_i| + \text{ATOL}_i} \ . \tag{9}$$

This permits an arbitrary combination of relative and absolute error control. The user-specified relative error tolerance is the scalar RTOL; the user-specified absolute error tolerance is ATOL, which may be a scalar or a vector. The value for RTOL indicates the number of digits of relative accuracy for a single time step. The specified value for ATOL_i indicates the values of the corresponding component of the solution vector which may be thought of as being zero, or at the noise level. In particular, if we set $\text{ATOL}_i = \text{RTOL} \times \text{FLOOR}_i$ then FLOOR_i represents the floor value for the ith component of the solution. The magnitude of FLOOR_i is the value for which there is a crossover from relative error control to absolute error control. In the case of vector absolute tolerances, a typical default for the scaled sensitivity vectors is to use the same ATOL as for the state variables y. Since the tolerances define the allowed error per step, they should be chosen conservatively. Experience indicates that a conservative choice yields a more economical solution than error tolerances that are too large.

For estimating the scaled sensitivity residual (4), SensIDA has an option that applies centered differences separately:

$$\frac{\partial F}{\partial y} w_i + \frac{\partial F}{\partial y'} w_i' \approx \frac{F(t, y + \delta_y w_i, y' + \delta_y w_i', p) - F(t, y - \delta_y w_i, y' - \delta_y w_i', p)}{2\delta_y} \tag{10}$$

and

$$\overline{p}_i \frac{\partial F}{\partial p_i} \approx \frac{F(t, y, y', p + \delta_i \overline{p}_i e_i) - F(t, y, y', p - \delta_i \overline{p}_i e_i)}{2\delta_i}. \tag{11}$$

As is typical for finite differences, the proper choice of perturbations δ_y and δ_i is a delicate matter. SensIDA uses a choice that takes into account several problem-related features: namely, the relative DAE error tolerance RTOL, the machine unit roundoff $\epsilon_{\text{machine}}$, and the weighted root-mean-square norm of the scaled sensitivity w_i. We then define

$$\delta_i = \sqrt{\max(\text{RTOL}, \epsilon_{\text{machine}})} \quad \text{and} \quad \delta_y = \frac{1}{\max(1/\delta_i, \|w_i\|_{rms})}. \tag{12}$$

The norm for $\|w_i\|$ uses the RTOL and ATOL_i associated with the state variables y. The terms $\epsilon_{\text{machine}}$ and $1/\delta_i$ are included as divide-by-zero safeguards in case RTOL $= 0$ or $\|w_i\| = 0$. Roughly speaking (i.e., if the safeguard terms are ignored), δ_i gives a $\sqrt{\text{RTOL}}$ relative perturbation to parameter i, and δ_y gives a unit weighted rms norm perturbation to y. Of course, the main drawback of this approach is that it requires four evaluations of $F(t, y, y', p)$.

Another centered differences technique for estimating the scaled sensitivity residuals uses

$$\frac{F(t, y + \delta w_i, y' + \delta w_i', p + \delta \overline{p}_i e_i) - F(t, y - \delta w_i, y' - \delta w_i', p - \delta \overline{p}_i e_i)}{2\delta} \tag{13}$$

in which

$$\delta = \min(\delta_i, \delta_y).$$

If $\delta_i = \delta_y$, a Taylor series analysis shows that the sum of (10) (11) and (13) are equivalent to within $O(\delta^2)$. The latter approach, however, is half as costly since it only requires two

evaluations of $F(t, y, y', p)$. To take advantage of this savings, it may also be desirable to use (13) when $\delta_i \approx \delta_y$. SensIDA accommodates this possibility by allowing the user to specify a threshold parameter $\rho_{\max}$. In particular, if δ_i and δ_y are within a factor of $|\rho_{\max}|$ of each other, then (13) is used to estimate the scaled sensitivity residuals. Otherwise, the sum of (10) (11) is used since δ_i and δ_y differ by a relatively large amount and the use of separate perturbations is prudent.

These procedures for choosing the perturbations (δ_i, δ_y, δ) and switching ($\rho_{\max}$) between centered difference formulas have also been implemented for first-order, forward difference formulas as well. In the latter case, forward differences can be applied separately or the single forward difference

$$\frac{\partial F}{\partial y} w_i + \frac{\partial F}{\partial y'} w_i' + \overline{p}_i \frac{\partial F}{\partial p_i} \approx \frac{F(t, y + \delta w_i, y' + \delta w_i', p + \delta \overline{p}_i e_i) - F(t, y, y', p)}{\delta} \tag{14}$$

can be used. In SensIDA, the default value of $\rho_{\max} = 0$ indicates the use of the centered difference (13) exclusively. Otherwise, the magnitude of $\rho_{\max}$ and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

In contrast to the above notation used in describing the mathematical details, the sections that follow use new variable names in explaining the organization, usage, and example programs of SensIDA. For convenient reference, we define these names as follows:

- `res` is the name of the user-supplied function that computes the DAE residual $F(t, y, y', p)$.
- `Ny` is the number of DAEs contained in `res` ($= N$ above)
- `Ns` is the number of sensitivity vectors w_i to be computed ($= m$ above)
- `Np` is the number of parameters contained in p ($\geq$ `Ns`)
- `Ntotal` is the total number of DAEs to be solved by SensIDA ($=$ `(1+Ns)Ny`)
- `yy` is the vector of length `Ntotal` that contains the `Ny` differential and algebraic variables and `Ns` scaled sensitivity vectors w_i
- `rhomax` is the finite difference threshold parameter $\rho_{\max}$
  For estimating the residual of the scaled sensitivity equations (4), the values and meanings for `rhomax` are as follows:
  - ◇ `rhomax = 0`: Use the centered difference (13).
  - ◇ `0 < rhomax < 1`: Use the sum of the centered differences (10) and (11).
  - ◇ `rhomax >= 1`: If δ_y and δ_i are within a factor of `rhomax` of each other, then (13) is used. Otherwise, the sum of the centered differences (10) and (11) is used.
  - ◇ `-1 < rhomax < 0`: Use the sum of (10) and (11), with forward differences instead of centered differences.
  - ◇ `rhomax <= -1`: If δ_y and δ_i are within a factor of `|rhomax|` of each other, then use (14). Otherwise, use the sum of (10) and (11) with forward differences instead of centered differences.

Lastly, SensIDA provides a way to compute the sensitivities of y with respect to a certain subset of the `Np` parameters. For instructions on specifying which `Ns` parameters are to be studied, see the description of `plist` in the next section.

3. Code Organization. One way to visualize SensIDA is to think of the code as being organized in layers, as partially shown in Fig. 1. The user's main program resides at the top level. The main program creates the required data structures, makes various initializations, defines the DAE residual `res`, defines the preconditioner setup and solve routines (if any), and makes calls to various modules at the second level. The main program also manages input/output.

At the second level, the `SENSIDA` module contains several routines that can be called by the user: `SensIDAMalloc`, for memory allocation and basic initializations related to sensitivity analysis; `SensIDAFree`, for memory deallocation; and `SensSetVecAtol`, for the vector case of setting absolute error tolerances for sensitivities. The `SENSIDA` routine `RESDQ` is called by the `IDA` module. `RESDQ` is responsible for computing the DAE residual by calling the user's `res` routine to evaluate $F(t, y, y', p)$ and for using various finite difference formulas to estimate the DAE scaled sensitivity residuals. By design, the `SENSIDA` module is independent of the choice of linear solver method used within the Newton iteration.

At the third level are the modules for the linear system solver, which at present are: `SensIDASPGMR`, `SensIDABAND`, `SensIDADENSE`. Each of these provides an interface to a corresponding generic solver for linear systems by a SPGMR, banded, or dense algorithm. In each case, the linear solver is called to solve the `(1+Ns)` linear systems of size `Ny`. The SPGMR method consists of the modules `SPGMR` and `ITERATIV`. `SensIDASPGMR` also accesses the user-supplied preconditioner solver routine, if specified, and possibly a user-supplied routine that computes and preprocesses the preconditioner by way of the Jacobian matrix or an approximation to it. The direct method modules, `SensIDABAND` and `SensIDADENSE`, access the user's Jacobian routine `jac` if one is supplied.

Finally, at the second level, the routine `IDASolve` within the `IDA` module is used to manage the time integration. `IDASolve` makes calls to `SENSIDA` to evaluate the scaled sensitivity residuals, and calls the linear solver module `SensIDADENSE`, `SensIDABAND`, or `SensIDASPGMR` to solve the linear systems that arise at each Newton iteration.

The following modules reside below the levels just described. The `LLNLTYPS` module defines types `real`, `integer`, and `boole` (boolean), and facilitates changing the precision of the arithmetic in the package from double to single, or the reverse. The `LLNLMATH` module specifies power functions and provides a function to compute the machine unit roundoff. Finally, we now describe the `NVECTOR` module.

In creating SensIDA from IDA, we developed a sensitivity version of the `NVECTOR` module. A revised `NVECTOR` module is needed because the overall DAE system has length `Ntotal`, but it consists of `1+Ns` DAE subsystems of length `Ny`; namely, the original nonlinear DAE system (1) and `Ns` scaled sensitivity residuals (4). Several steps are involved in partitioning and distributing the subsystems in a multiprocessor environment. First, each processor is responsible for solving a contiguous portion of each subsystem, of length `Nlocal`. Note that `Nlocal` need not be the same for each processor; however, the sum of all the `Nlocal` values must be `Ny`. Furthermore, the `1+Ns` subsystems of size `Ny` are identically partitioned among the processors. This implementation is handled through the revised `NVECTOR` module and its use of abstract data types: `type machEnvType`, for the machine environment data block (e.g., `Nlocal`); and `type N_Vector`, a data structure for the partitioned and distributed vectors

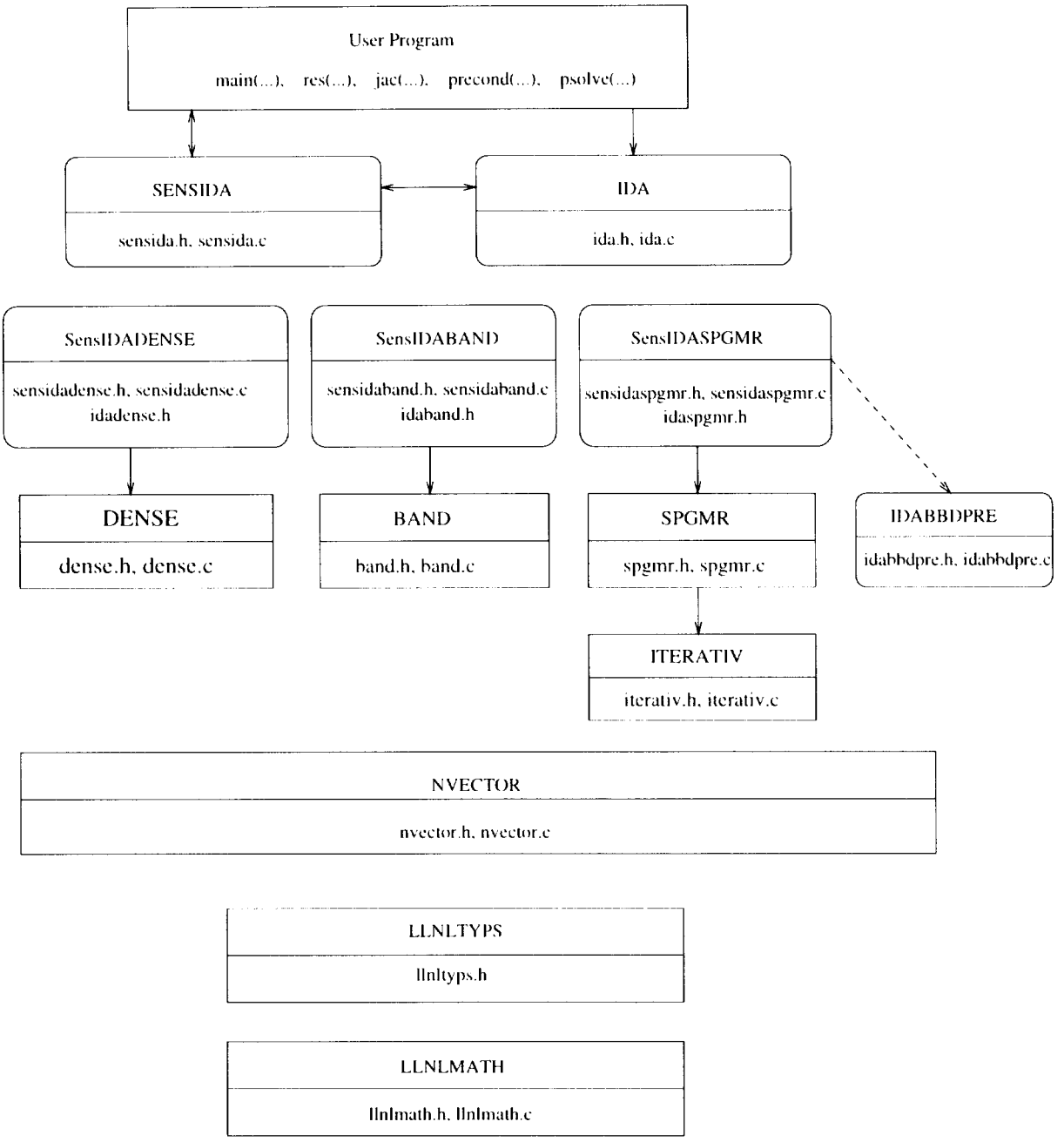

FIG. 1. *Overall structure of the SensIDA package. Modules comprising the central solver are distinguished by rounded boxes, while the user program, linear solver, and auxiliary modules are in unrounded boxes.*

just described. To achieve parallelism for any vector operation, each processor performs the operation on its assigned portions of the input vectors, followed by a global reduction where needed. In this way, vector calculations can be performed simultaneously with each processor working on its own block-portions of the vector.

The parallel version of SensIDA uses MPI (Message Passing Interface [8]) for all interprocessor communication. This achieves a high degree of portability, since MPI is becoming widely accepted as a standard for message passing software. For a different parallel computing environment, some rewriting of the vector module could allow the use of other specific machine-dependent instructions.

4. Using SensIDA. This section describes the use of SensIDA. We begin with a brief overview, in the form of a skeleton user program for parallel applications. Following that are detailed descriptions of the interface to the various user-callable routines, and of the user-supplied routines. We also describe a preconditioner module that is part of the IDA package. Finally, there are comments on usage under C++.

4.1. Overview of Usage. The following is a skeleton of the user's main program (or calling program) as an application of SensIDA on a parallel machine. The user program is to have these steps in the order indicated. In the serial case, Steps 3, 5, 17, and 18 can be omitted. For the sake of brevity, we defer many of the details to the later subsections, or to the IDA user document [9].

1. `#include` header files needed, to obtain various type definitions, enumerations, macros, etc. The files include `llnltyps.h`, `llnlmath.h`, `ida.h`, `nvector.h`, `mpi.h`; one or more of the files `idadense.h`, `idaband.h`, `idaspgmr.h`, `idabbdpre.h` associated with the choice of preconditioner and/or linear system solvers; and `sensida.h`. Also, the calling program must set the integer variables `Ny`, `Np`, `Ns` and `Ntotal`.
2. The calling program must define a pointer to a user-defined data block that is passed to the user's `res` routine. This data block must include a real pointer (for example, `p`) that points to the array of real parameters used by `res` to evaluate $F(t, y, y', p)$. For example, if the pointer to the data block has the form `typedef struct {..., real *p} *rdata`, then `rdata->p = p` must point to the real array in which `p[i-1]` = p_i, for `i=1,...,Np`.
3. `MPI_Init(&argc, &argv)` to initialize MPI if used by the user's program. Here `argc` and `argv` are the command line argument counter and array received by `main`.
4. Set `Nlocal` = the local vector length for this processor, and `Ny` = the global vector length for this processor. Note that `Ny` is the sum of all values of `Nlocal`.
5. `machEnv = PVecInitMPI(comm, Nlocal, Ny, &argc, &argv)` followed by `if (machEnv == NULL) return(1)`, to initialize the `NVECTOR` module. Here `comm` is the MPI communicator, which may be set by suitable MPI calls, for a proper subset of the active processors, or else set to either `NULL` or `MPI_COMM_WORLD` to specify that all processors are to be used.
6. The calling program must declare a real pointer (e.g., `pbar`) and set an array of real values `pbar[i]` that are used to scale the `Ns` sensitivity vectors w_i. Each `pbar[i]` must be set to a nonzero constant that is dimensionally consistent with

p[i]. Typically, pbar[i]=p[i] whenever p[i] is nonzero.

7. The pointer plist must be set to NULL or point to an array of integers. For the latter, this array can be allocated using plist = (int *) malloc(Ns * sizeof(int)). For the default setting plist = NULL, yysub[i] contains the scaled sensitivity of y with respect to p_i. Otherwise, the user must set the integers in plist to indicate which of the Ns sensitivity vectors are to be computed. Namely, yysub[i] will contain the scaled sensitivity of y with respect to p_j, where j=plist[i-1] and i=1,...,Ns.
8. Create and set N_Vector yy and N_Vector yp to initial values for Y and Y', respectively. Depending on user options, also create the vector id of differential/algebraic component flags and/or the vector constraints of inequality constraint flags. If an existing data array ydata contains the initial values of yy, then call yy = SensN_VMAKE(Ntotal, ydata, machEnv). Otherwise, make the call yy = N_VNew(Ntotal, machEnv) and load initial values into the array of size Ntotal defined by N_VDATA(yy).
   Conceptually, yy consists of (1+Ns) vectors of length Ny. If yy was created with N_VNew, then load the Ny initial values with N_VDATA(yy) = ydata where ydata is a data array of size Ny. To load the ith sensitivity vector, use yysub = N_VSUB(yy); N_VDATA(yysub[i]) = wdata, where wdata is an existing data array of length Ny. Note that yysub[0] is a pointer to the N_Vector of state variables y, and that yysub[i] is a pointer to the N_Vector for the ith scaled sensitivity vector w_i. Likewise, set yp to Y'.
9. If needed, several commands are available for identifying the components of the Ns sensitivity vectors as differential or algebraic variables, for initializing the sensitivity vectors to zero, and for setting the absolute error tolerances in the case of vector tolerances. To identify each component of the Ns sensitivity vectors as a differential or algebraic variable, use SensSetId(id, Ns), where the vector id is the differential/algebraic components flag used for yy in Step 7. The Ns sensitivity vectors can be initialized to zero using SensInitZero(yy, Ns). Finally, for the case of vector tolerances for absolute error control, a typical default is to use the same atol as for the state variables y. Use SensSetVecAtol(atol, Ns) to do this. Then, alter the resulting vector elements if desired.
10. ida_mem = SensIDAMalloc(...) allocates internal memory for IDA, initializes IDA, and returns a pointer to the IDA memory structure. (See details below.)
11. Specify the linear system solver to be used by making one of the calls:
    flag = SensIDADense(...) or flag = SensIDABand(...) or
    flag = SensIDASpgmr(...) followed by a test if (flag != 0) return(1).
12. Optionally, correct the initial values in yy, yp with the call
    flag = IDACalcIC(idamem, icopt, ...); if (flag != 0) return(1).
13. iflag = IDASolve(ida_mem, tout, yy, &t, itask) for each point t = tout at which output is desired. Set itask to NORMAL to have the integrator overshoot tout and interpolate, or ONE_STEP to take a single step and return. Alternatively, set itask = ONE_STEP to take one step forward and return. Also, test for the condition

flag < 0 to detect a failure. Following the call, process the vectors yy and yp, the computed solution Y and Y' at $t =$ tt. The *unscaled* sensitivity vector s_i is obtained by multiplying yysub[i] by the reciprocal of pbar[i]. To do this, call N_VScale(1.0/pbar[i], yysub[i], s_i).

14. SensIDAFree(ida_mem) to free the memory allocated for IDA.
15. The memory that was created for the vector yy in Step 7 must be deallocated: call N_VFree(yy) if yy was allocated by yy = N_VNew(...), or the user must call SensN_VDISPOSE(Ntotal, yy) if yy was allocated by yy = SensN_VMAKE(...).
16. Before freeing the pointer to the user-defined data block rdata, release the arrays containing the scale factors pbar and the real parameters p: free(pbar); free(rdata->p); free(rdata).
17. PVecFreeMPI(machEnv) to free machine-dependent data.
18. MPI_Finalize();

As indicated above, error conditions are possible at many of the steps, and are flagged by nonzero return values. In addition, error messages are issued in most cases.

The form of the call to SensIDAMalloc is

```
ida_mem = SensIDAMalloc(Ny, Ns, Ntotal, res, rdata, t0, y0, yp0,
                        itol, rtol, atol, id, constraints, errfp, optIn,
                        iopt[], ropt[], machEnv, p[], pbar[], plist, rhomax);
```

Except for a few additions, the arguments in SensIDAMalloc are the same as for the IDA routine IDAMalloc: the integer variables (Ny, Ns, Ntotal), the real pointers (p, pbar), and the integer pointer plist are described above; and the real variable rhomax is the finite difference threshold parameter ($\rho_{\max}$): see the description relating (13) to (14) in §2. res is the C function to compute $F(t, y, y', p)$. rdata is a pointer to the user-defined data block passed directly to the user's res function. t0 is the initial value of t. y0 is the vector of length Ntotal containing the initial values of Y (which can be the same as the vector yy described above), and yp0 is the vector of length Ntotal containing the initial value for the derivative Y'. The next three parameters are used to set the error control. The flag itol is replaced by either SS or SV, where SS indicates scalar relative error tolerance and scalar absolute error tolerance, while SV indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the DAE. The arguments &rtol and atol are pointers to the user's error tolerances. id is an N_Vector, required conditionally, which states a given element to be either algebraic or differential. constraints is an N_Vector defining inequality constraints for each component of the vector Y. The file pointer errfp points to the file where error messages from SensIDA are to be written (NULL for stdout). If optIn is replaced by FALSE, then the user is not going to provide optional input, while if it is TRUE then optional inputs are examined in iopt and ropt. iopt and ropt are integer and real arrays for optional input and output. machEnv is a pointer to machine environment-specific information. Full details for the arguments common to IDAMalloc can be found in [9].

4.2. User-Supplied Functions. The user-supplied routines consist of one function defining the DAE residual, and (optionally) one or two functions that define the precondi-

tioner for use in the SPGMR algorithm. All of the specifications are the same as when using IDA, as documented in [9]. However, recall that `Ny` refers to the number of DAEs contained in `res`, and the user-supplied data structure `rdata` contains a pointer (for example, `p`) that points to the array of real parameters used in `res`.

The function $F(t, y, y', p)$ defining the DAE system must be supplied by the user in the form of a C function, denoted `res` in the Overview of Usage.

```
int res(integer Ny, real tres, N_Vector yy, N_Vector yp,
        N_Vector resval, void *rdata)
```

This function takes as input the problem size `Ny`, the independent variable value `tres`, the dependent variable vector `yy`, and the derivative (with respect to t) of the `yy` vector, `yp`. The computed value of $F(t, y, y', p)$ is stored in `resval`.

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P is a left preconditioner matrix. Preconditioning is an important part of using IDA with the SPGMR solver (or any Krylov solver). In any nontrivial DAE problem, it is usually essential to provide a preconditioner of some sort. This is primarily because the Krylov iteration convergence test is based on the preconditioned residual vector. Without preconditioning, this test can be a very poor measure of convergence.

In supplying a preconditioner, the user must supply a C routine `PSolve` of the following form:

```
int PSolve(integer Ny, real tt, N_Vector yy, N_Vector yp,
       N_Vector rr, real cj, ResFn res, void *rdata, void *pdata,
       N_Vector ewt, real delta, N_Vector rvec, N_Vector zvec,
       long int *nrePtr, N_Vector tempv)
```

Its input includes `tt`, the current value of the independent variable; `yy`, the current value of the dependent variable vector y of length `Ny`; `yp`, the current value of the derivative vector y'; `rr`, the current vector of the residual vector $F(t, y, y', p)$; `cj`, the scalar in the system Jacobian, proportional to $1/h$; `res`, the residual function for the DAE problem; `rdata`, a pointer to user data to be passed to `res`; `pdata`, a pointer to user preconditioner data. Further input parameters are `ewt`, the input error weight vector; `delta`, an input tolerance if `Psolve` is to use an iterative method; `rvec`, the input right-hand side vector `r` in the linear system; `zvec`, the computed solution vector `z`; `nrePtr`, a pointer to the memory location containing the IDA problem data `nre` (the number of calls to `res`); `tempv`, a pointer to a memory location for temporary storage. The integer returned value is to be negative if the `Psolve` function failed with an nonrecoverable error, 0 if `Psolve` was successful, or positive if there was a recoverable error.

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then this needs to be done in the optional user-supplied C function `Precond`. The `Precond` function has the form:

```
int Precond(integer Ny, real tt, N_Vector yy, N_Vector yp,
      N_Vector rr, real cj, ResFn res, void *rdata, void *pdata,
      N_Vector ewt, N_Vector constraints, real hh, real uround,
      long int *nrePtr, N_Vector tempv1, N_Vector tempv2,
      N_Vector tempv3)
```

The arguments which have not been discussed previously are the following: `constraints`, the constraints vector; `hh`, a tentative step size in `t`; `uround`, is the machine unit roundoff; `tempv1`, `tempv2`, `tempv3` are temporary `N_vectors` available for workspace. The current stepsize `hh` and unit roundoff `uround` are supplied for possible use in difference quotient calculations.

4.3. Band-Block-Diagonal Preconditioner Module. SensIDA has the same `IBBDPRE` preconditioner module that is included in IDA. This preconditioner was developed to treat a rather broad class of problems based on solving partial differential equations (PDEs) using a method of lines approach. The modules generate a preconditioner that is a block-diagonal matrix, and each block contains a band matrix. The blocks need not have the same number of super- and sub-diagonals; these numbers may vary from block to block. The IDA user's guide [9] gives a complete description of this preconditioning technique. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate residual function. This requires the definition of a new function $G(t, y, y', p)$ which approximates the function $F(t, y, y', p)$ as given in (1). The choice $G = F$ is certainly allowed, but a less expensive choice may be just as effective for preconditioning.

To use this `IBBDPRE` module with SensIDA, the user must supply two functions which the module calls to construct the preconditioner matrix P. These are in addition to the user-supplied residual function `res`.

- A function `glocal(tt, yy, yp, gg, rdata)` must be supplied by the user to compute $G(t, y, y', p)$. It loads the vector `gg` as a function of `tt`, `yy`, `yp` and the parameters p contained in `rdata`. Although `yy`, `yp`, and `gg` are all of type `N_Vector`, only the local segment of each, of length `Nlocal`, is to be accessed in the routine `glocal`.
- A function `gcomm(yy, yp, rdata)` which must be supplied to perform all interprocessor communications necessary for the execution of the `glocal` function.

Both functions take as input the same pointer `rdata` as that passed by the user to `SensIDAMalloc` and passed to the user's function `res`. Both are to return an `int` equal to 0 (indicating success), or else 1 or -1 (indicating recoverable or non-recoverable failure, respectively), just as for `res`. The user is responsible for providing space (presumably within `*rdata`) for data that are communicated by `gcomm` from the other processors, and that are then used by `glocal`. The function `glocal` is not expected to do any communication.

The usage of the `IBBDPRE` module requires: (a) a call to `IBBDAlloc` to supply required parameters; and (b) passing specific names for the preconditioning routines in the call to `SensIDASpgmr`. See [9] for details.

4.4. Use by a C++ Application. SensIDA is written in a manner that permits it to be used by applications written in C++ as well as in C. For this purpose, each SensIDA header file is wrapped with conditionally compiled lines reading `extern "C" {...}`, conditional on the variable `__cplusplus` being defined. This directive causes the C++ compiler to use C-style names when compiling the function prototypes encountered. Users with C++ applications should also be aware that we have defined, in `llnltyps.h`, a boolean variable type, `boole`, since C has no such type. The type `boole` is equated to type `int`, and so arguments in user calls, or calls to user-supplied routines, which are of type `boole` can be

typed as either `boole` or `int` by the user. The same applies to vector kernels which have a type `boole` return value, if the user is providing these kernels. The name `boole` was chosen to avoid a conflict with the C++ type `bool`.

5. Example Problems. The SensIDA package includes eight sensitivity analysis example programs. These are based on three DAE system problems, two of which are solved in several different ways. The last of those two, a food web problem, is the most difficult and most realistic. Collectively, the examples are intended to illustrate the usage of both the serial and parallel versions of SensIDA, the usage of all three linear system modules, the use of the IDACalc routine for obtaining consistent initial conditions, and the use of the IDABBDPRE preconditioner module. In the following, we present the three DAE problems and describe how each is solved with SensIDA.

5.1. Robertson Kinetics Problem. This example, due to Robertson, is a model of a three-species chemical kinetics system written in DAE form. Differential equations are given for species y^1 and y^2 while an algebraic equation determines y^3. The equations for the system concentrations $y'(t)$ are:

$$\begin{cases} dy^1/dt = -p_1 y^1 + p_2 y^2 y^3 \\ dy^2/dt = p_1 y^1 - p_2 y^2 y^3 - p_3 (y^2)^2 \\ 0 = y^1 + y^2 + y^3 - 1 \end{cases} \tag{15}$$

The initial values are: $y^1 = 1.$, $y^2 = 0.$, and $y^3 = 0.$. The parameter values are: $p_1 = 0.04$, $p_2 = 10^4$, and $p_3 = 3 \times 10^7$. This example computes the three concentration components on the interval from $t = 0$ through 4×10^{10}.

This problem was solved only in one serial case using the simplest linear solver module, IDADENSE. It illustrates the application of IDADENSE, with a user-supplied Jacobian function, for those problems to which a dense solver is applicable.

The code and corresponding output can be found as `sensrobx.c` and `sensrobx.output` in the distributed package.

5.2. Heat Equation Problem. This example solves a discretized 2D heat PDE problem. The DAE system arises from the Dirichlet boundary condition $u = 0$, along with the differential equations arising from the discretization of the interior of the region.

The equations solved are:

$$\begin{cases} \partial u/\partial t = p_1 u_{xx} + p_2 u_{yy} & \text{(interior)} \\ u = 0. & \text{(boundary)}. \end{cases} \tag{16}$$

Initial conditions are given by $u = 16x(1-x)y(1-y)$, where the spatial domain is the unit square $0 \le x, y \le 1$. The parameter values are $p_1 = 1.0$, $p_2 = 1.0$, and the time interval is $0 \le t \le 10.24$.

We discretize this PDE system (16) (plus boundary conditions) with central differencing on a 10×10 mesh, so as to obtain a DAE system of size $N = 100$. The dependent variable vector u consists of the values $u^i(x_j, y_k, t)$ grouped first by x, and then by y. At each spatial boundary point, the boundary condition is coupled algebraically into the adjacent interior points by the central differencing scheme.

We solved this problem in four different ways, with the following example programs:

1. `sensheatsb`: serial version of SensIDA, band linear solver. The half-bandwidths are 10.
2. `sensheatsk`: serial version of SensIDA, Krylov (GMRES) linear solver with a user-supplied preconditioner. As a preconditioner, we use the diagonal elements of the matrix J.
3. `sensheatpk`: parallel version of SensIDA, Krylov (SPGMR) linear solver with a user-supplied preconditioner. We use a 5×5 subgrid on each of four processors. For the preconditioner, we again use the diagonal elements of the matrix J.
4. `sensheatbbd`: parallel version of SensIDA, Krylov (SPGMR) linear solver with `IDABBDPRE` preconditioner module. We use a 5×5 subgrid on each of 4 processors. We use half-bandwidths `mudq = mldq = 5` on each processor for the difference quotient scheme, but keep only a tridiagonal matrix (`mu = ml = 1`).

The source program for all four cases, along with the corresponding output files are available in the distributed package. They are not included in this document.

In the Appendix, we give the source and output of the `sensheatpk` program.

5.3. Food Web Problem. This example is a model of a multi-species food web [1], in which predator-prey relationships with diffusion in a 2D spatial domain are simulated. Here we consider a model with $s = 2p$ species: p predators and p prey. Species $1, \ldots, p$ (the prey) satisfy rate equations, while species $p+1, \ldots, s$ (the predators) have infinitely fast reaction rates. The coupled PDEs for the species concentrations $c^i(x, y, t)$ are:

$$\text{(17)} \qquad \begin{cases} \partial c^i/\partial t = R_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) \quad (i = 1, 2, \ldots, p) \ , \\ \qquad 0 = R_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) \quad (i = p+1, \ldots, s) \ , \end{cases}$$

with

$$R_i(x, y, c) = c^i(b_i + \sum_{j=1}^{s} a_{ij} c^j) \ .$$

Here c denotes the vector $\{c^i\}$. The interaction and diffusion coefficients (a_{ij}, b_i, d_i) can be functions of (x, y) in general. The choices made for this test problem are as follows, and include 2 parameters in the term b_i:

$$\begin{cases} a_{ii} = -1 \quad (\text{all } i) \\ a_{ij} = -0.5 \cdot 10^{-6} \quad (i \le p, j > p) \\ a_{ij} = 10^4 \ (i > p, j \le p) \\ (\text{all other } a_{ij} = 0) \ . \end{cases}$$

$$\begin{cases} b_i = b_i(x, y) = (1 + p_1 xy + p_2 \sin(4\pi x) \sin(4\pi y)) \quad (i \le p) \\ b_i = b_i(x, y) = -(1 + p_1 xy + p_2 \sin(4\pi x) \sin(4\pi y)) \quad (i > p) \ , \end{cases}$$

and

$$\begin{cases} d_i = 1 \quad (i \le p) \\ d_i = 0.5 \quad (i > p). \end{cases}$$

The spatial domain is the unit square $0 \le x, y \le 1$, and the time interval is $0 \le t \le 1$. The parameters values are: $p_1 = 50$, $p_2 = 1000$. The boundary conditions are of Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when $p_1 = p_2 = 0$. Empirically, a stable equilibrium appears to exist for (17) when p_1 and p_2 are positive, although it may not be unique. For the initial conditions, we set each prey concentration to a simple polynomial profile satisfying the boundary conditions, while the predator concentrations are all set to a flat value:

$$\begin{cases} c^i(x, y, 0) = 10 + i[16x(1-x)y(1-y)]^2 \quad (i \le p) \ , \\ c^i(x, y, 0) = 10^5 \quad (i > p) \ . \end{cases}$$

We discretize this PDE system (17) (plus boundary conditions) with central differencing on an $L \times L$ mesh, so as to obtain a DAE system of size $N = sL^2$. The dependent variable vector C consists of the values $c^i(x_j, y_k, t)$ grouped first by species index i, then by x, and lastly by y. At each spatial mesh point, the system has a block of p ODE's followed by a block of p algebraic equations, all coupled. For this example, we take $p = 1, s = 2$, and $L = 20$. See also [4], where various cases of this problem are solved with DASPK.

This problem was solved in three different ways, with the following three example programs:

1. `senswebsb`: serial version of SensIDA, band linear solver. The half-bandwidths are `mu = ml = sL = 40`.
2. `senswebpk`: parallel version of SensIDA, Krylov (SPGMR) linear solver with a user-supplied preconditioner. We use a $Lsub \times Lsub$ subgrid, with $Lsub = 10$, on each of four processors. For the preconditioner, we take the block-diagonal matrix with 2×2 blocks arising from the reaction coefficients $\partial R_i / \partial c$ only.
3. `senswebbbd`: parallel version of SensIDA, Krylov (SPGMR) linear solver with `IDABBDPRE` preconditioner module. We use half-bandwidths `mudq = mldq` $= sLsub =$ 20 for the difference quotient scheme, but retain only a matrix with bandwidth 5 by setting `mu = ml = 2`.

In all three cases, the flat predator initial values are not consistent with the quasi-steady equations for the predator species, and so we call `IDACalcIC` to correct those values. In the two parallel programs, we use a logically square array of processors and corresponding Cartesian subdomain decomposition.

6. Availability. The SensIDA package is being released for general distribution at this time. Interested users should contact Steven Lee (slee@llnl.gov) or Alan Hindmarsh (alanh@llnl.gov).

REFERENCES

[1] Peter N. Brown, *Decay to Uniform States in Food Webs*, SIAM J. Appl. Math., 46 (1986), pp. 376–392.

[2] Peter N. Brown and Alan C. Hindmarsh, *Reduced Storage Matrix Methods in Stiff ODE Systems*, J. Appl. Math. & Comp., **31** (1989), pp. 40-91.

[3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold, *Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems*, SIAM J. Sci. Comput., 15 (1994), pp. 1467–1488.

[4] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold, *Consistent Initial Condition Calculation for Differential-Algebraic Systems*, SIAM J. Sci. Comput., 19 (1998), pp. 1495–1512.

[5] George D. Byrne and Alan C. Hindmarsh, *User Documentation for PVODE, an ODE Solver for Parallel Computers*, Lawrence Livermore National Laboratory report UCRL-ID-130884, May 1998. See also the Addenda in the doc subdirectory of the current version of *PVODE*.

[6] Scott D. Cohen and Alan C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, Sept. 1994.

[7] Scott D. Cohen and Alan C. Hindmarsh, *CVODE, a Stiff/Nonstiff ODE Solver in C*, Computers in Physics, **10**, No. 2 (1996), pp. 138–143.

[8] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

[9] Alan C. Hindmarsh and Allan G. Taylor, *User Documentation for IDA, a Differential-Algebraic Equation Solver for Sequential and Parallel Computers*, Lawrence Livermore National Laboratory report UCRL-MA-136910, December 1999.

[10] Steven L. Lee, Alan C. Hindmarsh, and Peter N. Brown, *User Documentation for SensPVODE, a Variant of PVODE for Sensitivity Analysis*, Lawrence Livermore National Laboratory report UCRL-MA-140211, August 2000.

[11] Shengtai Li and Linda R. Petzold, *Software and Algorithms for Sensitivity Analysis of Large-Scale Differential Algebraic Systems*, To appear J. Comp. Appl. Math.

[12] Timothy Maly and Linda R. Petzold, *Numerical Methods and Software for Sensitivity Analysis of Differential-algebraic systems*, Appl. Numer. Math., **20** (1996), pp. 57–79.

[13] Yousef Saad and Martin Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., **11** (1990), 856-869.

7. Appendix: Listing of Heat Equation Example, with Sensitivity Analysis.

```
/*****************************************************************
 * File:       sensheatpk.c
 * Written by: Steven L. Lee and Alan C. Hindmarsh
 *---------------------------------------------------------------
 *
 * Sensitivity analysis version of Example problem for SensIDA:
 * 2D heat equation, parallel, GMRES.
 *
 * This example solves a discretized 2D heat equation problem.
 *
 *
 * The DAE system solved is a spatial discretization of the PDE
 *
 *          du/dt = p1*d^2u/dx^2 + p2*d^2u/dy^2
 *
 * on the unit square, where p1 = 1.0 and p2 = 1.0.
 * The boundary condition is u = 0 on all edges.
 * Initial conditions are given by u = 16 x (1 - x) y (1 - y).
 * The PDE is treated with central differences on a uniform MX x MY grid.
 * The values of u at the interior points satisfy ODEs, and equations
 * u = 0 at the boundaries are appended, to form a DAE system of size
 * N = MX * MY.  Here MX = MY = 10.
 *
 * The system is actually implemented on submeshes, processor by processor,
 * with an MXSUB by MYSUB mesh on each of NPEX * NPEY processors.
 *
 * The system is solved with SensIDA using the Krylov linear solver IDASPGMR.
 * The preconditioner uses the diagonal elements of the Jacobian only.
 * Routines for preconditioning, required by IDASPGMR, are supplied here.
 * The constraints u >= 0 are posed for all components.
 * Local error testing on the boundary values is suppressed.
 * Output is taken at t = 0, .01, .02, .04, ..., 10.24.
 *
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "llnltyps.h"
#include "llnlmath.h"
#include "nvector.h"
#include "ida.h"
```

```
#include "idaspgmr.h"
#include "iterativ.h"
#include "mpi.h"
#include "sensida.h"

#define ZERO  RCONST(0.0)
#define ONE   RCONST(1.0)
#define TWO   RCONST(2.0)


#define NOUT         11             /* Number of output times */

#define NPEX         2              /* No. PEs in x direction of PE array */
#define NPEY         2              /* No. PEs in y direction of PE array */
                                    /* Total no. PEs = NPEX*NPEY */
#define MXSUB        5              /* No. x points per subgrid */
#define MYSUB        5              /* No. y points per subgrid */

#define MX           (NPEX*MXSUB)   /* MX = number of x mesh points */
#define MY           (NPEY*MYSUB)   /* MY = number of y mesh points */
                                    /* Spatial mesh is MX by MY */
#define NY           (MX*MY)        /* number of equations      */
#define NP           2              /* number of parameters     */
#define NS           2              /* number of sensitivities  */

typedef struct {
  real      *p;
  integer   neq, thispe, mx, my, ixsub, jysub, npex, npey, mxsub, mysub;
  real      dx, dy, coeffx, coeffy, coeffxy;
  real      uext[(MXSUB+2)*(MYSUB+2)];
  N_Vector  pp;    /* vector of diagonal preconditioner elements */
  MPI_Comm  comm;
} *UserData;


/* Prototypes of private helper functions */

static int InitUserData(integer Neq, integer thispe, integer npes,
MPI_Comm comm, UserData data);

static int SetInitialProfile(N_Vector uu, N_Vector up, N_Vector id,
     N_Vector res, UserData data);
```

```
/* User-supplied residual function and supporting routines */


int heatres(integer Neq, real tres, N_Vector uu, N_Vector up,
            N_Vector res, void *rdata);

static int rescomm(N_Vector uu, N_Vector up, void *rdata);

static int reslocal(real tres, N_Vector uu, N_Vector up,
    N_Vector res,  void *rdata);

static int BSend(MPI_Comm comm, integer thispe, integer ixsub, integer jysub,
                 integer dsizex, integer dsizey, real uarray[]);

static int BRecvPost(MPI_Comm comm, MPI_Request request[], integer thispe,
     integer ixsub, integer jysub,
     integer dsizex, integer dsizey,
     real uext[], real buffer[]);

static int BRecvWait(MPI_Request request[], integer ixsub, integer jysub,
     integer dsizex, real uext[], real buffer[]);

/* User-supplied preconditioner routines */

int PSolveHeateq(integer local_N, real tt, N_Vector uu,
                 N_Vector up, N_Vector rr, real cj, ResFn res, void *rdata,
                 void *pdata, N_Vector ewt, real delta, N_Vector rvec,
                 N_Vector zvec, long int *nrePtr, N_Vector tempv);


int PrecondHeateq(integer local_N, real tt, N_Vector yy,
                  N_Vector yp, N_Vector rr, real cj,
                  ResFn res, void *rdata, void *pdata,
                  N_Vector ewt, N_Vector constraints, real hh,
                  real uround, long int *nrePtr,
                  N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);


main(int argc, char *argv[])

{
  integer retval, i, j, iout, itol, itask, local_N, npes, thispe;
```

```
long int iopt[OPT_SIZE];
boole optIn;
real ropt[OPT_SIZE], rtol, atol;
real t0, t1, tout, tret, umax;
void *mem;
UserData data;
N_Vector uu, up, constraints, id, res;
IDAMem idamem;
MPI_Comm comm;
machEnvType machEnv;
N_Vector *uusub, *upsub;
N_Vector *constraintssub;
integer Ntotal;
real *pbar;
real rhomax;
int *plist;

Ntotal = (1+NS)*NY;

/* Get processor number and total number of pe's. */
MPI_Init(&argc, &argv);
comm = MPI_COMM_WORLD;
MPI_Comm_size(comm, &npes);
MPI_Comm_rank(comm, &thispe);

if (npes != NPEX*NPEY) {
  if (thispe == 0)
    printf("\n npes=%d is not equal to NPEX*NPEY=%d\n", npes,NPEX*NPEY);
  return(1);
}

/* Set local length local_N. */
local_N = MXSUB*MYSUB;

/* Set machEnv block. */
machEnv = PVecInitMPI(comm, local_N, NY, &argc, &argv);
if (machEnv == NULL) return(1);

/* Allocate and initialize the data structure and N-vectors. */
data = (UserData) malloc(sizeof *data);

/* Store nominal parameter values in p */
data->p = (real *) malloc(NP * sizeof(real));
```

```
data->p[0] = 1.0;
data->p[1] = 1.0;

/* Scaling factor for each sensitivity equation */
pbar = (real *) malloc(NP * sizeof(real));
pbar[0] = 1.0;
pbar[1] = 1.0;

/* Store ordering of parameters in plist */
plist = (int *) malloc(NP * sizeof(int));
plist[0] = 1;
plist[1] = 2;

rhomax = 0.0;

uu = N_VNew(Ntotal, machEnv);
up = N_VNew(Ntotal, machEnv);
res = N_VNew(Ntotal, machEnv);
constraints = N_VNew(Ntotal, machEnv);
id = N_VNew(Ntotal, machEnv);

/* Create pointers to subvectors */
uusub = N_VSUB(uu);
upsub = N_VSUB(up);
constraintssub = N_VSUB(constraints);

data->pp = N_VNew(NY, machEnv); /* An N-vector to hold preconditioner. */

InitUserData(NY, thispe, npes, comm, data);

/* Initialize the uu, up, id, and res profiles. */
SetInitialProfile(uu, up, id, res, data);

/* Set constraints to all 1's for nonnegative solution values in y. */
N_VConst(ONE, constraintssub[0]);

/* Initialize the sensitivity variables */
SensInitZero(uu, NS);
SensInitZero(up, NS);
SensInitZero(constraints, NS);

/* Identify all sensitivity variables as differential variables */
SensSetId(id, NS);
```

```
t0 = 0.0; t1 = 0.01;

/* Scalar relative and absolute tolerance. */
itol = SS;
rtol = 0.0;
atol = 1.e-3;

/* Set option to suppress error testing of algebraic components. */
optIn = TRUE;
for (i = 0; i < OPT_SIZE; i++) {iopt[i] = 0; ropt[i] = ZERO; }
iopt[SUPPRESSALG] = 1;

/* Call SensIDAMalloc to initialize solution.  (NULL argument is errfp.) */
itask = NORMAL;

mem = SensIDAMalloc(NY, NS, Ntotal, heatres, data, t0, uu, up,
    itol, &rtol, &atol, id, constraints, NULL, optIn,
    iopt, ropt, machEnv, data->p, pbar, plist, rhomax);
if (mem == NULL) {
  if (thispe == 0) printf ("SensIDAMalloc failed.");
  return(1); }
idamem = (IDAMem)mem;

/* Call SensIDASpgmr to specify the linear solver. */
retval = SensIDASpgmr(idamem, PrecondHeateq, PSolveHeateq,  MODIFIED_GS,
                  0, 0, 0.0, 0.0, data);

if (retval != SUCCESS) {
  if (thispe == 0) printf("SensIDASpgmr failed, returning %d.\n",retval);
  return(1);
}

/* Call IDACalcIC (with default options) to correct the initial values. */
retval = IDACalcIC(idamem, CALC_YA_YDP_INIT, t1, ZERO, 0,0,0,0, ZERO);

if (retval != SUCCESS) {
  if (thispe == 0) printf("IDACalcIC failed. retval = %d\n", retval);
  return(1);
}

/* Compute the max norm of uu. */
umax = N_VMaxNorm(uusub[0]);
```

```
/* Print output heading (on processor 0 only). */

if (thispe == 0) {
  printf("sensheatpk: Heat equation, parallel example problem for SensIDA \n");
  printf("            Discretized heat equation on 2D unit square. \n");
  printf("            Zero boundary conditions,");
  printf(" polynomial initial conditions.\n");
  printf("            Mesh dimensions: %d x %d", MX, MY);
  printf("            Total system size: %d\n\n\n", NY);
  printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
  printf("            Processor array: %d x %d\n", NPEX, NPEY);
  printf("Number of sensitivities: Ns = %d\n", NS);
  printf("Parameter values:       p_1 = %9.2e,    p_2 = %9.2e\n",
 data->p[0], data->p[1]);
  printf("Scale factors:       pbar_1 = %9.2e, pbar_2 = %9.2e\n",
 pbar[0], pbar[1]);
  printf("Finite difference:   rhomax = %g\n", rhomax);
  printf("Tolerance parameters:  rtol = %g,  atol = %g\n", rtol, atol);
  printf("Constraints set to force all components of solution u >= 0. \n");
  printf("iopt[SUPPRESSALG] = 1 to suppress local error testing on");
  printf(" all boundary components. \n");
  printf("Linear solver: IDASPGMR  ");
  printf("Preconditioner: diagonal elements only.\n");

  /* Print output table heading and initial line of table. */
  printf("\n");
  printf("Output Summary:   max(u) = max-norm of solution u \n");
  printf("                max(s_i) = max-norm of sensitivity vector s_i\n\n");
  printf("  time       max(u)     k   nst  nni  nli   nre    h       npe nps\n");
  printf(" . . . . . . . . . . . . . . . . . . . . . . . . . . . .\n");

  printf(" %5.2f   %13.5e  %d  %3d  %3d  %3d  %4d %9.2e  %3d %3d\n",
         t0, umax, iopt[KUSED], iopt[NST], iopt[NNI], iopt[SPGMR_NLI],
         iopt[NRE], ropt[HUSED], iopt[SPGMR_NPE], iopt[SPGMR_NPS]);
}

for (i = 1; i <= NS; i++){
  umax = N_VMaxNorm(uusub[i]);
  j = (plist == NULL) ? i : plist[i-1];
  if (thispe == 0) {
    printf("max(s_%d) = %11.5e\n", j, umax/pbar[j-1]);
    if (i == NS) printf("\n");
```

```
    }
  }

  /* Loop over tout, call IDASolve, print output. */

  for (tout = t1, iout = 1; iout <= NOUT; iout++, tout *= TWO) {

    retval = IDASolve(idamem, tout, t0, &tret, uu, up, itask);

    umax = N_VMaxNorm(uusub[0]);
    if (thispe == 0)
printf(" %5.2f   %13.5e  %d  %3d  %3d  %3d  %4d %9.2e  %3d %3d\n",
         tret, umax,iopt[KUSED],iopt[NST],iopt[NNI],iopt[SPGMR_NLI],
               iopt[NRE], ropt[HUSED], iopt[SPGMR_NPE], iopt[SPGMR_NPS]);

    for (i = 1; i <= NS; i++){
      umax = N_VMaxNorm(uusub[i]);
      j = (plist == NULL) ? i: plist[i-1];
      if (thispe == 0) {
printf("max(s_%d) = %11.5e\n", j, umax/pbar[j-1]);
if (i == NS) printf("\n");
      }
    }

    if (retval < 0) {
      if (thispe == 0) printf("IDASolve returned %d.\n",retval);
      return(1);
    }

  }  /* End of tout loop. */

  /* Print remaining counters and free memory. */
  if (thispe == 0) printf("\n netf = %d,   ncfn = %d,   ncfl = %d \n",
                   iopt[NETF], iopt[NCFN], iopt[SPGMR_NCFL]);

  SensIDAFree(idamem);
  N_VFree(uu);
  N_VFree(up);
  N_VFree(constraints);
  N_VFree(id);
  N_VFree(res);
  N_VFree(data->pp);
```

```
  if (plist != NULL) free(plist);
  free(pbar);
  free(data->p);
  free(data);
  PVecFreeMPI(machEnv);
  MPI_Finalize();
  return(0);

} /* End of sensheatpk main program. */


/*********************************************************/
/* InitUserData initializes the user's data block data. */

static int InitUserData(integer Neq, integer thispe, integer npes,
        MPI_Comm comm, UserData data)
{

  data->neq = Neq;
  data->thispe = thispe;
  data->dx = ONE/(MX-ONE);       /* Assumes a [0,1] interval in x. */
  data->dy = ONE/(MY-ONE);       /* Assumes a [0,1] interval in y. */
  data->coeffx  = ONE/(data->dx * data->dx);
  data->coeffy  = ONE/(data->dy * data->dy);
  data->coeffxy = TWO/(data->dx * data->dx) + TWO/(data->dy * data->dy);
  data->jysub   = thispe/NPEX;
  data->ixsub   = thispe - data->jysub * NPEX;
  data->npex    = NPEX;
  data->npey    = NPEY;
  data->mx      = MX;
  data->my      = MY;
  data->mxsub = MXSUB;
  data->mysub = MYSUB;
  data->comm    = comm;
  return(0);

} /* End of InitUserData. */


/*************************************************************/
/* SetInitialProfile sets the initial values for the problem. */

static int SetInitialProfile(N_Vector uu, N_Vector up,  N_Vector id,
```

```
    N_Vector res, UserData data)
{
  integer i, iloc, j, jloc, offset, loc, ixsub, jysub;
  integer ixbegin, ixend, jybegin, jyend;
  real xfact, yfact, *udata, *iddata, dx, dy;

  /* Initialize uu. */

  udata = N_VDATA(uu);
  iddata = N_VDATA(id);

  /* Set mesh spacings and subgrid indices for this PE. */
  dx = data->dx;
  dy = data->dy;
  ixsub = data->ixsub;
  jysub = data->jysub;

  /* Set beginning and ending locations in the global array corresponding
     to the portion of that array assigned to this processor. */
  ixbegin = MXSUB*ixsub;
  ixend   = MXSUB*(ixsub+1) - 1;
  jybegin = MYSUB*jysub;
  jyend   = MYSUB*(jysub+1) - 1;

 /* Loop over the local array, computing the initial profile value.
    The global indices are (i,j) and the local indices are (iloc,jloc).
    Also set the id vector to zero for boundary points, one otherwise. */

  N_VConst(ONE, id);
  for (j = jybegin, jloc = 0; j <= jyend; j++, jloc++) {
    yfact = data->dy*j;
    offset= jloc*MXSUB;
    for (i = ixbegin, iloc = 0; i <= ixend; i++, iloc++) {
      xfact = data->dx * i;
      loc = offset + iloc;
      udata[loc] = 16. * xfact * (ONE - xfact) * yfact * (ONE - yfact);
      if (i == 0 || i == MX-1 || j == 0 || j == MY-1) iddata[loc] = ZERO;
      }
    }

  /* Initialize up. */

  N_VConst(ZERO, up);     /* Initially set up = 0. */
```

```
  /* heatres sets res to negative of ODE RHS values at interior points. */
  heatres(data->neq, ZERO, uu, up, res, data);

  /* Copy -res into up to get correct initial up values. */
  N_VScale(-ONE, res, up);

  return(SUCCESS);

} /* End of SetInitialProfiles. */


/***************** Functions called by the IDA solver ******************/


/*****************************************************************************
 * heatres: heat equation system residual function
 * This uses 5-point central differencing on the interior points, and
 * includes algebraic equations for the boundary values.
 * So for each interior point, the residual component has the form
 *    res_i = u'_i - (central difference)_i
 * while for each boundary point, it is res_i = u_i.
 *
 * This parallel implementation uses several supporting routines.
 * First a call is made to rescomm to do communication of subgrid boundary
 * data into array uext.  Then reslocal is called to compute the residual
 * on individual processors and their corresponding domains.  The routines
 * BSend, BRecvPost, and BREcvWait handle interprocessor communication
 * of uu required to calculate the residual. */

int heatres(integer Neq, real tres, N_Vector uu, N_Vector up,
            N_Vector res, void *rdata)
{
  int retval;
  UserData data;

  data = (UserData) rdata;

  /* Call rescomm to do inter-processor communication. */
  retval = rescomm(uu, up, data);

  /* Call reslocal to calculate res. */
  retval = reslocal(tres, uu, up, res, data);
```

```
  return(0);

} /* End of residual function heatres. */


/*  Supporting functions for heatres. */


/*********************************************************************/
/* rescomm routine.  This routine performs all inter-processor
   communication of data in u needed to calculate G.                 */

static int rescomm(N_Vector uu, N_Vector up, void *rdata)
{
  UserData data;
  real *uarray, *uext, buffer[2*MYSUB];
  MPI_Comm comm;
  integer thispe, ixsub, jysub, mxsub, mysub;
  MPI_Request request[4];

  data = (UserData) rdata;
  uarray = N_VDATA(uu);

  /* Get comm, thispe, subgrid indices, data sizes, extended array uext. */
  comm = data->comm;  thispe = data->thispe;
  ixsub = data->ixsub;   jysub = data->jysub;
  mxsub = data->mxsub;   mysub = data->mysub;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs. */
  BRecvPost(comm, request, thispe, ixsub, jysub, mxsub, mysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs. */
  BSend(comm, thispe, ixsub, jysub, mxsub, mysub, uarray);

  /* Finish receiving boundary data from neighboring PEs. */
  BRecvWait(request, ixsub, jysub, mxsub, uext, buffer);

  return(0);

} /* End of rescomm. */
```

```
/*********************************************************************/
/* reslocal routine.  Compute res = F(t, uu, up).  This routine assumes
   that all inter-processor communication of data needed to calculate F
    has already been done, and that this data is in the work array uext.  */

static int reslocal(real tres, N_Vector uu, N_Vector up, N_Vector res,
                    void *rdata)
{
  real *uext, *uuv, *upv, *resv;
  real termx, termy, termctr;
  integer i, lx, j, ly, offsetu, offsetue, locu, locue;
  integer ixsub, jysub, mxsub, mxsub2, mysub, npex, npey;
  integer ixbegin, ixend, jybegin, jyend;
  UserData data;
  real p1, p2;

  /* Get subgrid indices, array sizes, extended work array uext. */

  data = (UserData) rdata;
  p1 = data->p[0];
  p2 = data->p[1];
  uext = data->uext;
  uuv = N_VDATA(uu);
  upv = N_VDATA(up);
  resv = N_VDATA(res);
  ixsub = data->ixsub; jysub = data->jysub;
  mxsub = data->mxsub; mxsub2 = data->mxsub + 2;
  mysub = data->mysub; npex = data->npex; npey = data->npey;

  /* Initialize all elements of res to uu. This sets the boundary
     elements simply without indexing hassles. */

  N_VScale(ONE, uu, res);

  /* Copy local segment of u vector into the working extended array uext.
     This completes uext prior to the computation of the res vector.     */

  offsetu = 0;
  offsetue = mxsub2 + 1;
  for (ly = 0; ly < mysub; ly++) {
    for (lx = 0; lx < mxsub; lx++) uext[offsetue+lx] = uuv[offsetu+lx];
    offsetu = offsetu + mxsub;
```

```
    offsetue = offsetue + mxsub2;
  }

  /* Set loop limits for the interior of the local subgrid. */

  ixbegin = 0;
  ixend   = mxsub-1;
  jybegin = 0;
  jyend   = mysub-1;
  if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
  if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;

  /* Loop over all grid points in local subgrid. */

  for (ly = jybegin; ly <=jyend; ly++) {
    for (lx = ixbegin; lx <= ixend; lx++) {
      locu  = lx + ly*mxsub;
      locue = (lx+1) + (ly+1)*mxsub2;
      termx = p1 * data->coeffx *(uext[locue-1]      + uext[locue+1]);
      termy = p2 * data->coeffy *(uext[locue-mxsub2] + uext[locue+mxsub2]);
      termctr = (p1*(data->coeffx) + p2*(data->coeffy)) * TWO * uext[locue];
      resv[locu] = upv[locu] - (termx + termy - termctr);
   }
  }
  return(0);

} /* End of reslocal. */


/*********************************************************************/
/* Routine to send boundary data to neighboring PEs.                 */

static int BSend(MPI_Comm comm, integer thispe, integer ixsub, integer jysub,
                 integer dsizex, integer dsizey, real uarray[])
{
  integer ly, offsetu;
  real bufleft[MYSUB], bufright[MYSUB];

  /* If jysub > 0, send data from bottom x-line of u. */

  if (jysub != 0)
    MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, thispe-NPEX, 0, comm);
```

```
  /* If jysub < NPEY-1, send data from top x-line of u. */

  if (jysub != NPEY-1) {
    offsetu = (MYSUB-1)*dsizex;
    MPI_Send(&uarray[offsetu], dsizex, PVEC_REAL_MPI_TYPE,
           thispe+NPEX, 0, comm);
  }

  /* If ixsub > 0, send data from left y-line of u (via bufleft). */

  if (ixsub != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = ly*dsizex;
      bufleft[ly] = uarray[offsetu];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, thispe-1, 0, comm);
  }

  /* If ixsub < NPEX-1, send data from right y-line of u (via bufright). */

  if (ixsub != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = ly*MXSUB + (MXSUB-1);
      bufright[ly] = uarray[offsetu];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, thispe+1, 0, comm);
  }
} /* End of BSend. */


/*************************************************************/
/* Routine to start receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*MYSUB real entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in
   both calls also. */

static int BRecvPost(MPI_Comm comm, MPI_Request request[], integer thispe,
     integer ixsub, integer jysub,
     integer dsizex, integer dsizey,
     real uext[], real buffer[])
```

```
{
  integer offsetue;
  /* Have bufleft and bufright use the same buffer. */
  real *bufleft = buffer, *bufright = buffer+MYSUB;

  /* If jysub > 0, receive data for bottom x-line of uext. */
  if (jysub != 0)
    MPI_Irecv(&uext[1], dsizex, PVEC_REAL_MPI_TYPE,
      thispe-NPEX, 0, comm, &request[0]);

  /* If jysub < NPEY-1, receive data for top x-line of uext. */
  if (jysub != NPEY-1) {
    offsetue = (1 + (MYSUB+1)*(MXSUB+2));
    MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
                                        thispe+NPEX, 0, comm, &request[1]);
  }

  /* If ixsub > 0, receive data for left y-line of uext (via bufleft). */
  if (ixsub != 0) {
    MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
                                        thispe-1, 0, comm, &request[2]);
  }

 /* If ixsub < NPEX-1, receive data for right y-line of uext (via bufright). */
  if (ixsub != NPEX-1) {
    MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
                                        thispe+1, 0, comm, &request[3]);
  }

} /* End of BRecvPost. */


/*************************************************************/
/* Routine to finish receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*MYSUB real entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have four entries, and should be passed in both
      calls also. */

static int BRecvWait(MPI_Request request[], integer ixsub, integer jysub,
      integer dsizex, real uext[], real buffer[])
```

```
{
  integer ly, dsizex2, offsetue;
  real *bufleft = buffer, *bufright = buffer+MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2;

  /* If jysub > 0, receive data for bottom x-line of uext. */
  if (jysub != 0)
    MPI_Wait(&request[0],&status);

  /* If jysub < NPEY-1, receive data for top x-line of uext. */
  if (jysub != NPEY-1)
    MPI_Wait(&request[1],&status);

  /* If ixsub > 0, receive data for left y-line of uext (via bufleft). */
  if (ixsub != 0) {
    MPI_Wait(&request[2],&status);

    /* Copy the buffer to uext. */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetue = (ly+1)*dsizex2;
      uext[offsetue] = bufleft[ly];
    }
  }

  /* If ixsub < NPEX-1, receive data for right y-line of uext (via bufright). */
  if (ixsub != NPEX-1) {
    MPI_Wait(&request[3],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetue = (ly+2)*dsizex2 - 1;
      uext[offsetue] = bufright[ly];
    }
  }
} /* End of BRecvWait. */


/*********************************************************************
 * PrecondHeateq: setup for diagonal preconditioner for heatsk.      *
 *                                                                   *
 * The optional user-supplied functions PrecondHeateq and            *
```

```
 * PSolveHeateq together must define the left preconditoner        *
 * matrix P approximating the system Jacobian matrix               *
 *                   J = dF/du + cj*dF/du'                         *
 * (where the DAE system is F(t,u,u') = 0), and solve the linear   *
 * systems P z = r.   This is done in this case by keeping only    *
 * the diagonal elements of the J matrix above, storing them as    *
 * inverses in a vector pp, when computed in PrecondHeateq, for    *
 * subsequent use in PSolveHeateq.                                 *
 *                                                                 *
 * In this instance, only cj and data (user data structure, with   *
 * pp etc.) are used from the PrecondHeateq argument list.         *
 *******************************************************************/

int PrecondHeateq(integer local_N, real tt, N_Vector yy,
                  N_Vector yp, N_Vector rr, real cj,
                  ResFn res, void *rdata, void *pdata,
                  N_Vector ewt, N_Vector constraints, real hh,
                  real uround, long int *nrePtr,
                  N_Vector tempv1, N_Vector tempv2, N_Vector tempv3)
{
  integer i, j, offset, loc;
  real *rv, *zv, *ppv, pelinv, pel;
  integer lx, ly, ixbegin, ixend, jybegin, jyend, locu, mxsub, mysub;
  integer ixsub, jysub, npex, npey;
  UserData data;
  real p1, p2;

  data = (UserData) pdata;
  p1 = data->p[0];
  p2 = data->p[1];

  ppv = N_VDATA(data->pp);
  ixsub = data->ixsub;
  jysub = data->jysub;
  mxsub = data->mxsub;
  mysub = data->mysub;
  npex  = data->npex;
  npey  = data->npey;

  /* Initially set all pp elements to one. */
  N_VConst(ONE, data->pp);

  /* Prepare to loop over subgrid. */
```

```
  ixbegin = 0;
  ixend   = mxsub-1;
  jybegin = 0;
  jyend   = mysub-1;
  if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
  if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
  pel = cj + ((p1*data->coeffx) + (p2*data->coeffy))*TWO;
  pelinv = ONE/pel;

  /* Load the inverse of the preconditioner diagonal elements
     in loop over all the local subgrid.                          */

  for (ly = jybegin; ly <=jyend; ly++) {
    for (lx = ixbegin; lx <= ixend; lx++) {
      locu  = lx + ly*mxsub;
      ppv[locu] = pelinv;
    }
  }

  return(SUCCESS);

} /* End of PrecondHeateq. */


/*****************************************************************
 * PSolveHeateq: solve preconditioner linear system.             *
 * This routine multiplies the input vector rvec by the vector pp *
 * containing the inverse diagonal Jacobian elements (previously  *
 * computed in PrecondHeateq), returning the result in zvec.     */

int PSolveHeateq(integer local_N, real tt, N_Vector uu,
                 N_Vector up, N_Vector rr, real cj, ResFn res, void *rdata,
                 void *pdata, N_Vector ewt, real delta, N_Vector rvec,
                 N_Vector zvec, long int *nrePtr, N_Vector tempv)
 {
  UserData data;

  data = (UserData) pdata;

  N_VProd(data->pp, rvec, zvec);

  return(SUCCESS);
```

```
} /* End of PSolveHeateq. */
```

Sample output for the example program sensheatpk.

```
sensheatpk: Heat equation, parallel example problem for SensIDA
            Discretized heat equation on 2D unit square.
            Zero boundary conditions, polynomial initial conditions.
            Mesh dimensions: 10 x 10           Total system size: 100

Subgrid dimensions: 5 x 5           Processor array: 2 x 2
Number of sensitivities: Ns = 2
Parameter values:       p_1 =  1.00e+00,    p_2 =  1.00e+00
Scale factors:       pbar_1 =  1.00e+00, pbar_2 =  1.00e+00
Finite difference:   rhomax = 0
Tolerance parameters:  rtol = 0,  atol = 0.001
Constraints set to force all components of solution u >= 0.
iopt[SUPPRESSALG] = 1 to suppress local error testing on all boundary components.
Linear solver: IDASPGMR  Preconditioner: diagonal elements only.

Output Summary:   max(u) = max-norm of solution u
                max(s_i) = max-norm of sensitivity vector s_i

  time       max(u)     k   nst  nni  nli   nre    h       npe nps
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  0.00     9.75461e-01  0    0    1    2     5  1.00e-05    2  11
max(s_1) = 0.00000e+00
max(s_2) = 0.00000e+00

  0.01     8.24106e-01  2   11   14   25    41  2.56e-03   10  73
max(s_1) = 7.20774e-02
max(s_2) = 7.20774e-02

  0.02     6.88134e-01  3   14   18   40    60  5.12e-03   10 100
max(s_1) = 1.27129e-01
max(s_2) = 1.27129e-01

  0.04     4.70846e-01  3   18   22   58    82  5.12e-03   10 130
max(s_1) = 1.81792e-01
max(s_2) = 1.81792e-01

  0.08     2.16343e-01  3   22   27   94   123  1.02e-02   11 181
max(s_1) = 1.68497e-01
max(s_2) = 1.68494e-01
```

```
  0.16      4.54871e-02  4   30    36  147    185  1.02e-02    11 261
max(s_1) = 7.05468e-02
max(s_2) = 7.05478e-02

  0.32      2.00938e-03  2   38    47  226    275  4.10e-02    13 373
max(s_1) = 5.95024e-03
max(s_2) = 5.94397e-03

  0.64      2.04003e-04  1   44    56  270    328  1.47e-01    15 444
max(s_1) = 4.18633e-04
max(s_2) = 4.18349e-04

  1.28      3.24684e-04  1   47    62  283    347  2.95e-01    19 475
max(s_1) = 3.74875e-04
max(s_2) = 3.79646e-04

  2.56      3.16884e-04  1   49    65  293    360  5.90e-01    20 494
max(s_1) = 5.97687e-04
max(s_2) = 5.87800e-04

  5.12      4.80339e-05  1   51    68  299    369  2.36e+00    22 509
max(s_1) = 4.89528e-04
max(s_2) = 4.51301e-04

 10.24      3.54902e-04  1   52    70  307    379  4.72e+00    23 523
max(s_1) = 1.84619e-04
max(s_2) = 1.57882e-04


 netf = 0,   ncfn = 1,   ncfl = 0
```